# Metaheurísticas

Problema del Aprendizaje de Pesos en Características.

**Práctica 2: Técnicas de Búsqueda basadas en Poblaciones.**


**UNIVERSIDAD DE GRANADA**

**E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN**

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial


**Grado en Ingeniería Informática. Tercero.**
**Curso 2018-2019.**

Algoritmos Genéticos y Meméticos.


*Daniel Bolaños Martínez*
*76592621-E*
*danibolanos@correo.ugr.es*
*Grupo 1 - Martes 17:30h*

# Índice

## 1. Descripción del Problema

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos.

La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así se puede formular como:

$$Maximizar\ F(W) = \alpha \cdot tasa\_clas(W) + (1 - \alpha) \cdot tasa\_red(W)$$

Partimos de un vector de datos $T = \{t_1, \ldots, t_n\}$ donde cada dato contiene un conjunto de características o *traits* $\{x_1, \ldots, x_n\}$.

El problema se reduce en calcular un vector de pesos $W = \{w_1, \ldots, w_n\}$ donde el peso $w_i \in [0, 1]$ pondera la característica $x_i$. A este vector lo denominaremos vector de entrenamiento o aprendizaje y nos permitirá clasificar otro conjunto de datos desconocido al que llamaremos vector de test o validación.

Para aumentar la fiabilidad del proceso usaremos el método *5-fold cross validation* que consiste en crear cinco particiones distintas de datos repartidos equitativamente según su clase. Las particiones las dividiremos de tal forma que dediquemos una a *test* y cuatro a *train*.

Para clasificar los datos, implementamos el algoritmo **k-NN** en su versión con $k = 1$. Este algoritmo asigna a cada dato la clase su vecino más cercano calculado a partir de la *distancia euclídea*.

Calcularemos el vector de pesos con los diferentes algoritmos y usaremos nuestro clasificador **k-NN** para valorar el rendimiento de los resultados. Además es necesario calcular los diferentes porcentajes para cada conjunto de datos valorados. Definiremos:

$$tasa\_clas = 100 \cdot \frac{n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0.2}{n^o\ caracteristicas}$$

Para proceder a los cálculos debemos partir de unos datos normalizados. En la primera práctica, implementamos un algoritmo **Greedy (Relief)** y uno de **Búsqueda Local** y en la segunda hemos desarrollado cuatro algoritmos **Genéticos** y tres **Meméticos**. Los algoritmos son ejecutados sobre un conjunto de datos que contienen los archivos *colposcopy*, *ionosphere*, *texture* todos ellos en el formato elegido *.csv*.

## 2. Descripción de los Algoritmos empleados en el Problema

Nuestro esquema de representación de la solución, será un vector de pesos $W$ que nos servirá para valorar la bondad de los datos en clasificaciones futuras. El vector será del tipo $W = \{w_1, \ldots, w_n\}$ donde cada $w_i \in [0, 1]$ y cada $w_i$ gradúa el peso asociado a cada característica y pondera su importancia.

Nuestro algoritmo **k-NN** medirá la bondad de los pesos calculados mediante cada algoritmo, recibiendo como parámetros un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculado por el algoritmo seleccionado. El clasificador devolverá un *struct Resultados* que contiene las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo que ha tardado en ejecutarse.

La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. A mayor tasa de clasificación, mejor será el vector de pesos $W$ generado por nuestro algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto al total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

El pseudocódigo del algoritmo **k-NN** es el siguiente:

```
function KNN_LOO(train, test, w)
   for i=0, i < tamaño test, i++ do
      pos = nearestNeighbour_LOO(train, test[i], w)
      if train[pos].clase = test[i].clase then
          aciertos++
      end if
   end for
   for i=0, i < tamaño w, i++ do
     if w[i] < 0.2 then
       num_w_menor++
     end if
   end for
  tasa_clas = 100.0*(aciertos / tamaño test)
  tasa_red = 100.0*(num_w_menor / tamaño w)
end function
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que estemos valorando. Además, ha sido programado aplicando leave-one-out, ya que cuando se utiliza en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto, porque estamos aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo dato que estamos comparando.

```
function nearestNeighbour_LOO(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1
   for i=0, i < tamaño train, i++ do
      // leave-one-out
      if actual != train[i] then
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
      end if
   end for
end function
```

Finalmente utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de características teniendo en cuenta los pesos cuyo valor supere 0.2.

```
function euclideanDistance(v1, v2, w)
   for i=0, i < tamaño v1, i++ do
      if w[i] >= 0.2 then
         dist = dist + w[i]*(v2[i]-v1[i])*(v2[i]-v1[i])
      end if
   end for
end function
```

Además, he añadido una versión de la función **k-NN** que utiliza una modalidad del *nearestNeighbour* sin el criterio de Leave-One-Out. Esta versión del algoritmo será usada cuando apliquemos el **k-NN** sobre dos conjuntos de datos diferentes.

```
function nearestNeighbour(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1
   for i=0, i < tamaño train, i++ do
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
   end for
end function
```

En los algoritmos genéticos y meméticos he creado un struct *Cromosoma* que contiene la información básica de cada elemento con el que vamos a trabajar (la lista de características y su valor de evaluación).

```
struct Cromosoma{
    vector<double> w;
```

```
    double pts;
};
```

Para seleccionar el conjunto de cromosomas padres de la población, usaremos la función *binaryTournament*, la cual genera dos números aleatorios, dentro de los índices de cada población de cromosomas y se queda con el que tenga una mejor puntuación de evaluación de los dos.

```
function binaryTournament(poblacion, generador)
  max = 0
  distribucion random_int(0, poblacion.size()-1)

  num1 = random_int(generador)
  while num1 = num2 do
    num2 = random_int(generador)
  end while

  if poblacion[num1].pts > poblacion[num2].pts then
    max = num1
  else
    max = num2
  end if

  return max
end function
```

Aplicando el torneo binario, obtendremos los padres de cada población que serán cruzados entre ellos para obtener los hijos de la nueva población. Usaremos dos operadores de cruce: *cruceBLX-*$\alpha$ y *cruceAritmetico*.

- El **cruce *BLX-*$\alpha$** con $\alpha = 0.3$, genera una pareja de descendientes a partir de dos padres, asignando a cada componente $w[i]$ de cada hijo un valor aleatorio dentro del rango: $(w_{min} - (w_{max} - w_{min}) \cdot \alpha, w_{max} + (w_{max} - w_{min}) \cdot \alpha)$ y truncando su valor entre 0 y 1.

```
function cruceBLX(padre1, padre2, generador)
  crear hijo1
  crear hijo2

  for i=0, i < tamaño caracteristicas padre1, i++ do
    max = maximo(padre1.w[i], padre2.w[i])
    min = minimo(padre1.w[i], padre2.w[i])
    diff = max - min
    distribucion random(min-diff*ALPHA_AGG, max+diff*ALPHA_AGG)

    aux = random(generador)
    truncar aux
    hijo1.w[i] = aux
```

```
    aux = random(generador)
    truncar aux
    hijo2.w[i] = aux
  end for

  return pair(hijo1, hijo2)
end function
```

- El **cruce Aritmético**, genera solo un descendiente que tiene como valor de cada componente $w[i]$ del vector de características, la media aritmética del valor de $w[i]$ de cada padre.

```
function cruceArit(padre1, padre2)
  crear hijo

  for i=0, i < tamaño características padre1, i++ do
    hijo.w[i] = (padre1.w[i]+padre2.w[i]) / 2.0
  end for

  return hijo
end function
```

Para realizar las mutaciones oportunas en los algoritmos genéticos y meméticos, realizaremos en cada generación el número de mutaciones esperadas sobre la población de hijos. Elegiremos un cromosoma y un gen aleatorios de la población y le aplicaremos una mutación sumándole un valor de una distribución normal con $\sigma = 0.3$ y finalmente truncando su valor.

```
for i=0, i < num_mutaciones, i++ do
  valor_mut = random_mutaciones(generador)
  cromosoma_mutar = valor_mut / tamaño caracteristicas
  gen_mutar = valor_mut % tamaño características
  aux = hijos[cromosoma_mutar].w[gen_mutar] + normal(generador)
  truncar aux
  hijos[cromosoma_mutar].w[gen_mutar] = aux
end for
```

La búsqueda local implementada en los algoritmos meméticos es similar a la utilizada en la práctica 1, salvo el criterio de parada e inicialización.

```
function BL_MEM(rain, cromosoma, num_eval)
  w = cromosoma.w
  index = {0,...,w.size()}
  mezcla los valores de index
  // Reutiliza la evaluación del cromosoma
  agr_ant = cromosoma.pts;

  // Condición de parada
  while neighbour < 2*tamaño w do
```

```
      aux = index[iter % tamaño w]
      w_mut = w
      w_mut[aux] += normal(generador)
      truncar el vector de pesos mutado
      // Evalúa el vector de pesos mutado
      agr_new = agregado(resultados KNN_LOO)
      iter++
      num_eval++
      if agr_new > agr_ant then
        w = w_mut
        agr_ant = agr_new
      end if
      neighbour++
      if iter % tamaño w == 0 then
        mezcla los valores de index
      end if
  end while

  h.pts = agr_ant
end function
```

Además he creado algunas funciones auxiliares para obtener el mejor y peor cromosoma dada una población, así como una para obtener una lista de los índices de los cromosomas ordenados de mejor a peor valor de evaluación.

```
function getBestCromosoma(poblacion)
  max = 0
  pts_max = 0.0

  for i=0, i < tamaño población, i++ do
    if poblacion[i].pts > pts_max then
      max = i
      pts_max = poblacion[i].pts
    end if
  end for

  return max
end function
```

//————————————————————//

```
function getWorstCromosoma(poblacion)
  min = 0
  pts_min = 100.0

  for i=0, i < tamaño población, i++ do
    if poblacion[i].pts < pts_min then
      min = i
```

```
      pts_min = poblacion[i].pts
    end if
  end for

  return min
end function
```

//————————————————————//

```
function getListBestCromosoma(poblacion)
  crear index(tamaño poblacion)

  for i=0, i < tamaño poblacion, i++ do
    index[i] = i
  end for

  ordenar index de mayor a menor poblacion[i].pts

  return index
end function
```

## 3. Pseudocódigo de los Algoritmos

### 3.1. Algoritmo Greedy (Relief)

El algoritmo se basa en incrementar el peso de aquellas características que mejor separan a ejemplos que son enemigos entre sí y reducir el valor del peso en aquellas características que separan ejemplos que son amigos entre sí.

El pseudocódigo del algoritmo **Greedy (Relief)** es el siguiente:

```
function Relief(train, w)
   w = {0,...,0}
   for i=0, i < tamaño train, i++ do
      nearestFriendEnemy(train, train[i], friend, enemy)
      for j=0, j < tamaño w, j++ do
         w[j]= w[j]+|train[i].t[j]-train[i].t[enemy]|
               -|train[i].t[j]-train[i].t[friend]|
      end for
   end for
   w_max = máximo del vector de pesos w
   for i=0, i < tamaño w, i++ do
      if w[i] < 0 then
         w[i] = 0
      else
         w[i] = w[i] / w_max
      end if
```

```
   end for
end function
```

La función *nearestFriendEnemy* calcula el amigo y el enemigo más cercano a un dato dado en función de la *distancia euclídea*. Un dato se considera enemigo si tiene la clase distinta y amigo si tiene la misma clase del dato actual. La función emplea leave-one-out para evitar comparar distancias con el dato actual.

```
function nearestFriendEnemy(train, actual, friend, enemy)
   for i=0, i < tamaño train, i++ do
      if actual != train[i] then
         distancia_actual = euclideanDistance(train[i].t, actual.t)
         if train[i].category = actual.category then
            if distancia_actual < mejor_distancia_a then
               mejor_distancia_a = distancia_actual
               friend = i
            end if
         end if
      else
         if distancia_actual < mejor_distancia_e then
            mejor_distancia_e = distancia_actual
            enemy = i
         end if
      end if
   end for
end function
```

### 3.2. Búsqueda Local

La búsqueda local implementa una búsqueda de primero el mejor. El vector index nos indica en qué orden se van a modificar las componentes. Modificando así en cada paso una componente aleatoria que no hayamos modificado antes. El vector de pesos $W$ se generará de manera aleatoria con valores entre $[0, 1]$ utilizando una distribución uniforme real.

Para poder generar soluciones nuevas, deberemos modificar/mutar el vector $W$ añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza $\sigma^2$, pero este método puede proporcionar soluciones negativas, por lo que debemos de truncar los valores negativos a 0.

El pseudocódigo del algoritmo **Búsqueda Local** es el siguiente:

```
function BL(train, w)
   w = distribucion_uniforme(0,1)
   index = {0,...,w.size()}

   mezcla los valores de index
```

```
   // clasifica el vector de pesos w con KNN y
   // calcula su agregado como tasa de evaluación
   antiguo = KNN(train, train, w)
   agr_ant = agregado(antiguo.clas,antiguo.red)

   while iter < MAX_ITER and neighbour < tamaño w*MAX_NEIGHBOUR do
      aux = index[iter % tamaño w]
      w_mut = w
      w_mut[aux] += normal(generator)
      truncar el vector de pesos mutado
      //clasifica el vector de pesos mutado
      // y calcula su agregado
      agr_new = agregado(resultados KNN_LOO)
      iter++
      if agr_new > agr_ant then
         w = w_mut
         agr_ant = agr_new
         neighbour = 0
      else
         neighbour++
      end if
      if iter % tamaño w = 0 then
         mezcla los valores de index
      end if
   end while
end function
```

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso y utiliza este resultado para evaluar la bondad de la solución obtenida. En este caso usamos alpha=0.5.

```
function agregado(t_clas, t_red)
  alpha*t_clas+(1.0-alpha)*t_red
end function
```

### 3.3. Algoritmos Genéticos Generacionales

- Partimos de una población de tamaño 30.
- Usaremos tres variables para indicar el mejor de la población anterior y el mejor y peor de la nueva población.
- Haremos las mutaciones esperadas (2 por generación) sobre el vector de hijos y evaluaremos con el **k-NN** los cromosomas que hayan mutado o hayan sido generados nuevamente.

- Aplicaremos *Elitismo* cuando el mejor cromosoma de la población anterior tenga un valor de evaluación superior al mejor de la nueva población. En ese caso, cambiaremos el mejor de la población anterior por el peor de la nueva.

### 3.3.1. Algoritmo Genético Generacional cruce BLX-$\alpha$

```
function AGG_BLX(train, w)
  // vector<Cromosoma> con TAM_PBL=30
  poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
  // número de evaluaciones a 0
  t = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = num_cruces * tamaño características
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)

  for i=0, i < TAM_PBL, i++ do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts > poblacion[mejor].pts
      mejor = i
    end if
    // Incremento el número de evaluaciones
    t++
  end for

  // MAX_ITER=15000
  while t < MAX_ITER do
    // Seleccionamos los padres con el torneo binario
    for i=0, i < TAM_PBL, i++ do
      select = binaryTournament(poblacion, generador)
      padres[i] = poblacion[select]
    end for

    // Hacemos el cruce BLX para obtener los hijos
    for i=0, i < num_cruces, i+=2 do
      hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
      poblacion_intermedia[i] = hijosBLX.first
      poblacion_intermedia[i+1] = hijosBLX.second
    end for

    // Hacemos el número de mutaciones esperadas
```

```
    for i=0, i < num_mutaciones, i++ do
      proceso de mutacion sobre poblacion_intermedia
    end for

    // Introducimos los últimos padres en la población
    for i=num_cruces, i < TAM_PBL, i++ do
      poblacion_intermedia[i] = padres[i]
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < num_cruces, i++ do
      poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
      t++
    end for

    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)

    // Elitismo
    if poblacion[mejor].pts > poblacion_intermedia[mejorNueva].pts then
      poblacion_intermedia[peorNueva] = poblacion[mejor]
      mejor = peorNueva
    else
      mejor = mejorNueva
    end if

    // Sustituimos la población
    poblacion = poblacion_intermedia
  end while

  w = poblacion[mejor].w
end function
```

### 3.3.2. Algoritmo Genético Generacional cruce Aritmético

Para el cruce Aritmético, como el operador de cruce; en este caso, solo devuelve un único descendiente, crearemos el doble de padres para realizar el doble de cruces y obtener hijos diferentes y finalmente añadiremos a la población nueva los padres restantes para completar (TAM_PBL-num_cruces).

```
function AGG_CA(train, w)
  // vector<Cromosoma> con TAM_PBL=30
  poblacion(TAM_PBL), padres(num_cruces+TAM_PBL), poblacion_intermedia(TAM_PBL)
  // número de evaluaciones a 0
  t = 0
```

```
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion random_mutaciones(0, genes_por_generacion-1)

for i=0, i < TAM_PBL, i++ do
  inicializar poblacion[i].w con distribucion uniforme
  // Evaluar cada cromosoma con el KNN_LOO
  poblacion[i].pts = agregado(resultados KNN_LOO)
  if poblacion[i].pts > poblacion[mejor].pts
    mejor = i
  end if
  // Incremento el número de evaluaciones
  t++
end for

// MAX_ITER=15000
while t < MAX_ITER do
  // Seleccionamos los padres con el torneo binario
  for i=0, i < num_cruces+TAM_PBL, i++ do
    select = binaryTournament(poblacion, generador)
    padres[i] = poblacion[select]
  end for

  // Hacemos el cruce Aritmético para obtener los hijos
  for i=0, i < num_cruces*2, i+=2 do
    hijoCA = cruceArit(padres[i], padres[i+1], generador)
    poblacion_intermedia[i/2] = hijoCA
  end for

  // Hacemos el número de mutaciones esperadas
  for i=0, i < num_mutaciones, i++ do
    proceso de mutacion sobre poblacion_intermedia
  end for

  // Introducimos los últimos padres en la población
  for i=num_cruces*2, i < num_cruces+TAM_PBL, i++ do
    poblacion_intermedia[i-num_cruces] = padres[i]
  end for

  // Evaluamos los nuevos hijos
  for i=0, i < num_cruces, i++ do
    poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
    t++
  end for
```

```
    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)

    // Elitismo
    if poblacion[mejor].pts > poblacion_intermedia[mejorNueva].pts then
      poblacion_intermedia[peorNueva] = poblacion[mejor]
      mejor = peorNueva
    else
      mejor = mejorNueva
    end if

    // Sustituimos la población
    poblacion = poblacion_intermedia
  end while

  w = poblacion[mejor].w
end function
```

### 3.4. Algoritmos Genéticos Estacionarios

- Partimos de una población de tamaño 30.
- En esta versión solo se generan dos hijos eligiendo en cada caso el cruce que corresponda.
- Usaremos tres variables para indicar el peor de la población anterior y el mejor y peor de la nueva población.
- Haremos las mutaciones esperadas (2 por generación) sobre el vector de hijos y evaluaremos con el **k-NN** los cromosomas que hayan mutado o hayan sido generados nuevamente.
- No se aplica *Elitismo* puesto que los hijos generados compiten con los dos peores de la población nueva para ser sustituidos.

#### 3.4.1. Algoritmo Genético Estacionario cruce BLX-$\alpha$

```
function AGE_BLX(train, w)
  // vector<Cromosoma> con TAM_PBL=30
  poblacion(TAM_PBL), poblacion_intermedia(2)
  // número de evaluaciones a 0
  t = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = 2 * tamaño características
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)
```

```
for i=0, i < TAM_PBL, i++ do
  inicializar poblacion[i].w con distribucion uniforme
  // Evaluar cada cromosoma con el KNN_LOO
  poblacion[i].pts = agregado(resultados KNN_LOO)
  if poblacion[i].pts < poblacion[peor].pts
    peor = i
  end if
  // Incremento el número de evaluaciones
  t++
end for

// MAX_ITER=15000
while t < MAX_ITER do
  // Seleccionamos los padres con el torneo binario
  select = binaryTournament(poblacion, generador)
  p1 = poblacion[select]
  select = binaryTournament(poblacion, generador)
  p2 = poblacion[select]

  // Hacemos el cruce BLX para obtener los hijos
  hijosBLX = cruceBLX(p1, p2, generador)
  poblacion_intermedia[0] = hijosBLX.first
  poblacion_intermedia[1] = hijosBLX.second

  // Hacemos el número de mutaciones esperadas
  for i=0, i < num_mutaciones, i++ do
    proceso de mutacion sobre poblacion_intermedia
  end for

  // Evaluamos los nuevos hijos
  for i=0, i < 2, i++ do
    poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
    t++
  end for

  if poblacion_intermedia[0].pts > poblacion_intermedia[1].pts then
    mejorNueva = 0
    peorNueva = 1
  else
    mejorNueva = 1
    peorNueva = 0
  end if

  if poblacion_intermedia[mejorNueva].pts > poblacion[peor].pts then
    poblacion[peor] = poblacion_intermedia[mejorNueva]
```

```
      peor = getWorstCromosoma(poblacion)
      if poblacion_intermedia[peorNueva].pts > poblacion[peor].pts then
        poblacion[peor] = poblacion_intermedia[peorNueva]
        peor = getWorstCromosoma(poblacion)
      end if
    end if
  end while

  w = poblacion[getBestCromosoma(poblacion)].w
end function
```

### 3.4.2. Algoritmo Genético Estacionario cruce Aritmético

De forma similar a su versión Generacional, en esta versión seleccionaremos el doble de padres para obtener los dos hijos de la nueva población.

```
function AGE_CA(train, w)
  // vector<Cromosoma> con TAM_PBL=30
  poblacion(TAM_PBL), poblacion_intermedia(2)
  // número de evaluaciones a 0
  t = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = 2 * tamaño características
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)

  for i=0, i < TAM_PBL, i++ do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts < poblacion[peor].pts
      peor = i
    end if
    // Incremento el número de evaluaciones
    t++
  end for

  // MAX_ITER=15000
  while t < MAX_ITER do
    // Seleccionamos los padres con el torneo binario
    select = binaryTournament(poblacion, generador)
    p1 = poblacion[select]
    select = binaryTournament(poblacion, generador)
    p2 = poblacion[select]
```

```
    // Hacemos el cruce Aritmetico para obtener los hijos
    poblacion_intermedia[0] = cruceArit(p1, p2)

    // Repetimos el proceso para el otro hijo
    select = binaryTournament(poblacion, generador)
    p1 = poblacion[select]
    select = binaryTournament(poblacion, generador)
    p2 = poblacion[select]

    poblacion_intermedia[1] = cruceArit(p1, p2)

    // Hacemos el número de mutaciones esperadas
    for i=0, i < num_mutaciones, i++ do
      proceso de mutacion sobre poblacion_intermedia
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < 2, i++ do
      poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
      t++
    end for

    if poblacion_intermedia[0].pts > poblacion_intermedia[1].pts then
      mejorNueva = 0
      peorNueva = 1
    else
      mejorNueva = 1
      peorNueva = 0
    end if

    if poblacion_intermedia[mejorNueva].pts > poblacion[peor].pts then
      poblacion[peor] = poblacion_intermedia[mejorNueva]
      peor = getWorstCromosoma(poblacion)
      if poblacion_intermedia[peorNueva].pts > poblacion[peor].pts then
        poblacion[peor] = poblacion_intermedia[peorNueva]
        peor = getWorstCromosoma(poblacion)
      end if
    end if
  end while

  w = poblacion[getBestCromosoma(poblacion)].w
end function
```

### 3.5. Algoritmos Meméticos

Para la implementación, usaremos una hibridación del Algoritmo Genético Generacional con el cruce que mejores resultados ha aportado (BLX-$\alpha$ en nuestro caso), con la Búsqueda Local. * La población se reduce a 10 individuos. * Usaremos una versión del *Búsqueda Local* que incluye una condición de parada hasta $2 \cdot num\_caracteristicas$ vecinos y que elimina la inicialización del vector de pesos, ya que usaremos el que pasemos por parámetro en cada caso. * El resto de indicaciones se mantiene como en el AGG_BLX.

#### 3.5.1. Algoritmo Memético BLX (10, 1.0)

Cada 10 generaciones, se aplica la Búsqueda Local sobre todos los cromosomas de la población.

```
function AM_BLX_10_1_0(train, w)
  // vector<Cromosoma> con TAM_PBL=10
  poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
  // número de evaluaciones a 0
  t = 0
  generacion = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = num_cruces * tamaño características
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)

  for i=0, i < TAM_PBL, i++ do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts > poblacion[mejor].pts
      mejor = i
    end if
    // Incremento el número de evaluaciones
    t++
  end for

  // MAX_ITER=15000
  while t < MAX_ITER do
    generacion++
    // Seleccionamos los padres con el torneo binario
    for i=0, i < TAM_PBL, i++ do
      select = binaryTournament(poblacion, generador)
      padres[i] = poblacion[select]
    end for
```

```
    // Hacemos el cruce BLX para obtener los hijos
    for i=0, i < num_cruces, i+=2 do
      hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
      poblacion_intermedia[i] = hijosBLX.first
      poblacion_intermedia[i+1] = hijosBLX.second
    end for

    // Hacemos el número de mutaciones esperadas
    for i=0, i < num_mutaciones, i++ do
      proceso de mutacion sobre poblacion_intermedia
    end for

    // Introducimos los últimos padres en la población
    for i=num_cruces, i < TAM_PBL, i++ do
      poblacion_intermedia[i] = padres[i]
    end for

    // Aplicamos Local Search a toda la población
    if generacion % 10 == 0 then
      for i=0, i < TAM_PBL, i++ do
        BL_MEM(train, poblacion_intermedia[i], t)
      end for
    end if

    // El BL_MEM se encarga de evaluar toda la población

    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)

    // Elitismo
    if poblacion[mejor].pts > poblacion_intermedia[mejorNueva].pts then
      poblacion_intermedia[peorNueva] = poblacion[mejor]
      mejor = peorNueva
    else
      mejor = mejorNueva
    end if

    // Sustituimos la población
    poblacion = poblacion_intermedia
  end while

  w = poblacion[mejor].w
end function
```

### 3.5.2. Algoritmo Memético BLX (10, 0.1)

Cada 10 generaciones, se aplica la Búsqueda Local sobre un subconjunto de cromosomas de la población, seleccionado aleatoriamente con probabilidad 0.1 para cada cromosoma.

```
function AM_BLX_10_0_1(train, w)
  // vector<Cromosoma> con TAM_PBL=10
  poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
  // número de evaluaciones a 0
  t = 0
  generacion = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = num_cruces * tamaño características
  num_bl_cromosoma = ceil(0.1*TAM_PBL)
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)

  for i=0, i < TAM_PBL, i++ do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts > poblacion[mejor].pts
      mejor = i
    end if
    // Incremento el número de evaluaciones
    t++
  end for

  // MAX_ITER=15000
  while t < MAX_ITER do
    generacion++
    // Seleccionamos los padres con el torneo binario
    for i=0, i < TAM_PBL, i++ do
      select = binaryTournament(poblacion, generador)
      padres[i] = poblacion[select]
    end for

    // Hacemos el cruce BLX para obtener los hijos
    for i=0, i < num_cruces, i+=2 do
      hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
      poblacion_intermedia[i] = hijosBLX.first
      poblacion_intermedia[i+1] = hijosBLX.second
    end for

    // Hacemos el número de mutaciones esperadas
```

```
    for i=0, i < num_mutaciones, i++ do
      proceso de mutacion sobre poblacion_intermedia
    end for

    // Introducimos los últimos padres en la población
    for i=num_cruces, i < TAM_PBL, i++ do
      poblacion_intermedia[i] = padres[i]
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < num_cruces, i++ do
      poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
      t++
    end for

    // Aplicamos Local Search a un cromosoma aleatorio de la población
    if generacion % 10 == 0 then
      for int i=0, i < num_bl_cromosoma, i++ do
        select = random_BL(generador)
        BL_MEM(train, poblacion_intermedia[select], t)
      end for
    end if

    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)

    // Elitismo
    if poblacion[mejor].pts > poblacion_intermedia[mejorNueva].pts then
      poblacion_intermedia[peorNueva] = poblacion[mejor]
      mejor = peorNueva
    else
      mejor = mejorNueva
    end if

    // Sustituimos la población
    poblacion = poblacion_intermedia
  end while

  w = poblacion[mejor].w
end function
```

### 3.5.3. Algoritmo Memético BLX (10, 0.1) mejor

Cada 10 generaciones, se aplica la Búsqueda Local sobre los $0.1 \cdot tam\_poblacion$

mejores cromosomas de la población actual.

```
function AM_BLX_10_0_1_mej(train, w)
  // vector<Cromosoma> con TAM_PBL=10
  poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
  // número de evaluaciones a 0
  t = 0
  generacion = 0
  // PB_MUT=0.7
  num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
  genes_por_generacion = num_cruces * tamaño características
  num_bl_cromosoma = ceil(0.1*TAM_PBL)
  num_mutaciones = ceil(PB_MUT * genes_por_generacion)
  distribucion random_mutaciones(0, genes_por_generacion-1)

  for i=0, i < TAM_PBL, i++ do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts > poblacion[mejor].pts
      mejor = i
    end if
    // Incremento el número de evaluaciones
    t++
  end for

  // MAX_ITER=15000
  while t < MAX_ITER do
    generacion++
    // Seleccionamos los padres con el torneo binario
    for i=0, i < TAM_PBL, i++ do
      select = binaryTournament(poblacion, generador)
      padres[i] = poblacion[select]
    end for

    // Hacemos el cruce BLX para obtener los hijos
    for i=0, i < num_cruces, i+=2 do
      hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
      poblacion_intermedia[i] = hijosBLX.first
      poblacion_intermedia[i+1] = hijosBLX.second
    end for

    // Hacemos el número de mutaciones esperadas
    for i=0, i < num_mutaciones, i++ do
      proceso de mutacion sobre poblacion_intermedia
    end for
```

```
    // Introducimos los últimos padres en la población
    for i=num_cruces, i < TAM_PBL, i++ do
      poblacion_intermedia[i] = padres[i]
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < num_cruces, i++ do
      poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
      t++
    end for

    lista = getListBestCromosoma(poblacion_intermedia)

    // Aplicamos Local Search al mejor cromosoma de la población
    if generacion % 10 == 0 then
      for int i=0, i < num_bl_cromosoma, i++ do
        select = lista[i]
        BL_MEM(train, poblacion_intermedia[select], t)
      end for
    end if

    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)

    // Elitismo
    if poblacion[mejor].pts > poblacion_intermedia[mejorNueva].pts then
      poblacion_intermedia[peorNueva] = poblacion[mejor]
      mejor = peorNueva
    else
      mejor = mejorNueva
    end if

    // Sustituimos la población
    poblacion = poblacion_intermedia
  end while

  w = poblacion[mejor].w
end function
```

## 4. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar los datos obtenidos para cada algoritmo es siempre el mismo y se realiza en la función *ejecutar* para cada algoritmo programado.

Primero obtenemos los pesos usando el algoritmo a comparar y seguidamente clasificamos con el **k-NN** los datos de train y test sobre ese vector de pesos obtenido. Por último devolvemos en un *struct Resultados* las tasas de clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

```
function ejecutarAlgoritmo(particion, i)
   test = toma la partición i
   train = toma la suma de datos de las particiones !=
   w = Algoritmo(train, w)
   resultados = KNN(train, test, w)
end function
```

## 5. Procedimiento del Desarrollo de la Práctica y Manual de Usuario

La práctica ha sido programada en el lenguaje C++. Para el desarrollo de la práctica, primero he necesitado leer los archivos que nos proporcionan los datos, para ello, he optado por pasar los archivos .arff a .csv, un formato que al menos para mí, es más manipulable. Seguidamente y con ayuda de la función *read_csv*, guardo los datos en memoria en un vector de estructuras que he denominado *FicheroCSV*. Cada *FicheroCSV* contiene la información de cada línea del fichero *csv* original incluyendo un vector de los datos y un string que indica la clase de los mismos. Por tanto, consideraremos cada conjunto de datos como un *vector<FicheroCSV>*.

Seguidamente, y para aplicar la *5-fold cross validation*, creo un vector con 5 particiones que contienen punteros (para reducir el coste de memoria) a estructuras *FicheroCSV* repartidas de forma equitativa. Los datos se introducen en la partición ya normalizados.

Una vez que tenemos el vector<vector<FicheroCSV*>>, ya podemos repartir los datos en train y test y proceder como hemos descrito anteriormente con los algoritmos programados.

Para la primera práctica, desarrollé una versión del **k-NN** sin leave-one-out, seguidamente programé el algoritmo **Greedy (Relief)** y finalmente la **Búsqueda Local** con la cual tuve que modificar parte del código del **k-NN** para que aplicase el leave-one-out.

Para la segunda práctica, primero programé el código para los cruces BLX y

Aritmético y más tarde desarrollé los 4 algoritmos **Genéticos**. Por últimos, cambié el código del **Búsqueda Local** para adaptarlo a las restricciones que se imponen en el guión y poder usarlo en los 3 algoritmos **Meméticos** que se nos piden.

En el archivo *LEEME.txt* se encuentra explicado el proceso para replicar las ejecuciones del programa. Así que me limitaré a resumirlo:

Existe un make que genera el ejecutable del programa. El programa ejecutable ha sido compilado con g++ y optimizado con -O2 para reducir los tiempos de ejecución.

./bin/p2 ./data/archivo.csv seed

Ejecuta los diez algoritmos programados en ambas prácticas sobre el conjunto de datos contenido en el *archivo.csv* (con $archivo = \{colposcopy, ionosphere, texture\}$) usando como semilla el número *seed* pasado como parámetro en el **Búsqueda Local**, **Algoritmos Genéticos** y **Meméticos**.

./bin/p2 seed

Crea un archivo *tablas_seed.csv* con los resultados de ejecutar los tres algoritmos implementados sobre los tres conjuntos de datos (*colposcopy.csv*, *ionosphere.csv*, *texture.csv*) y vuelca su contenido en una tabla. El parámetro *seed* indica la iniciacización de la semilla.

Tanto los ficheros .csv como el fichero tablas_seed.csv que se genere se podrán encontrar en el directorio ./data.

## 6. Experimentos y Análisis de Resultados

Para la obtención de los resultados de esta práctica, hemos utilizado los siguientes conjuntos de datos:

- **Colposcopy**: Conjunto de datos de secuencias colposcópicas extraídas a partir de las características de las imágenes. Contiene 287 ejemplos con 62 características cada uno clasificados en 2 categorías.
- **Ionosphere**: Conjunto de datos de radar que indican el número de electrones libres en la ionosfera partir de las señales procesadas. Contiene 352 ejemplos con 34 características cada uno clasificados en 2 categorías.
- **Texture**: Conjunto de datos para distinguir entre 11 tipos de texturas diferentes a partir de imágenes. Contiene 550 ejemplos con 40 características cada uno clasificados en 11 categorías.

El algoritmo de **Búsqueda Local** y los **Genéticos** y **Meméticos** dependen de un parámetro que especifica la semilla para le generación de números aleatorios. Voy a utilizar para el análisis de estos resultados SEED=3. Los resultados se guardarán por tanto en el archivo *tablas_3.csv* del directorio *data*.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza $\sigma^2$ donde $\sigma = 0.3$. Como criterio de parada en la **Búsqueda Local**, se va a usar el número de evaluaciones que se han realizado así como el número vecinos por característica explorados. En nuestro caso los valores serán 15000 y $20 \cdot tam\_vector\_pesos$ respectivamente.

Las tablas se generan automáticamente con la ejecución del programa y contienen los resultados para cada partición, así como la media de los resultados para cada algoritmo.

A continuación, se muestran las tablas para cada fichero y cada algoritmo utilizado:

| | COLPOSCOPY | | IONOSPHERE | | TEXTURE | |
|---|---|---|---|---|---|---|
| | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red |
| **1-NN** | 74,9062 | 0 | 86,5996 | 0 | 92,5455 | 0 |
| **Relief** | 74,9062 | 40,9677 | 87,4567 | 2,94118 | 93,0909 | 5,5 |
| **BL** | 72,4864 | 80,3226 | 88,0241 | 88,2353 | 87,2727 | 84,5 |
| **AGG_BLX** | 72,4924 | 77,4194 | 86,33 | 85,2941 | 90,1818 | 79 |
| **AGG_CA** | 73,5148 | 64,8387 | 84,6197 | 73,5294 | 90,3636 | 73 |
| **AGE_BLX** | 74,9183 | 84,5161 | 85,1952 | 90 | 90 | 85,5 |
| **AGE_CA** | 73,5088 | 80,9677 | 86,3219 | 87,6471 | 90,3636 | 85 |
| **AM(1.0)** | 73,8596 | 81,2903 | 84,6157 | 89,4118 | 89,4545 | 86,5 |
| **AM(0.1)** | 74,8941 | 78,7097 | 83,4769 | 89,4118 | 88,5455 | 84,5 |
| **AM(0.1)mej** | 77,7314 | 78,7097 | 87,1871 | 90 | 89,6364 | 86,5 |

Figure 1: Tabla datos comparativa.csv

Teniendo en cuenta solo los algoritmos de la primera práctica, podemos ver que los mejores resultados son obtenidos por el algoritmo de **Greedy-Relief**, igualando al **1-NN** para el conjunto de datos *colposcopy*. Mientras que el **Búsqueda Local** es el que obtiene en los tres casos la mejor tasa de reducción, lo que indica que se necesitan menos atributos para clasificar los datos con respecto al resto de algoritmos.

Nos quedaremos por tanto con el **Búsqueda Local** para hacer las comparaciones con los algoritmos evolutivos.

Respecto a la segunda práctica, las tasas de clasificación de los diferentes algoritmos son similares en media, distan solo en 5% como mucho. Siendo mejores los algoritmos meméticos consistentemente.

Si observamos la tasa de reducción, en este caso, podemos ver que el **Búsqueda Local** obtiene bastante buenos resultados de media, solo es superado por los **Genéticos Estacionarios** y los **Meméticos** salvo excepciones.

Esto es debido a que en la **Búsqueda Local** empezamos muy pronto a descartar características para poder clasificar los datos, en los algoritmos genéticos hay que esperar a varias mutaciones para que al final se acabe realizando un proceso similar al de la **Búsqueda Local**. Es por ello por lo que obtiene en varias

ocasiones; resultados mejores, sobre todo en conjuntos de datos más pequeños, en los que los algoritmos evolutivos no explotan todo su potencial.

Visualizando los resultados de *ionosphere.csv* (el que menor cantidad de datos presenta) podemos ver que el algoritmos **BL** obtiene la mejor tasa de clasificación, mientras que no por ello presenta una mejor tasa de reducción. Sobre *colposcopy.csv*, ya no presenta los mejores resultados de clasificación, por lo que podemos concluir con que para una cantidad de datos $x \in (11.968, 17.794]$ los algoritmos evolutivos comienzan a ser mejores.

Podemos ver los resultados totales obtenidos para cada conjunto de datos:

| | COLPOSCOPY | | | |
|---|---|---|---|---|
| | 1-NN | | | |
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| **Partición 1** | 81,0345 | 0 | 40,5172 | 0,003552 |
| **Partición 2** | 70,6897 | 0 | 35,3448 | 0,001941 |
| **Partición 3** | 77,193 | 0 | 38,5965 | 0,00149 |
| **Partición 4** | 68,4211 | 0 | 34,2105 | 0,00115 |
| **Partición 5** | 77,193 | 0 | 38,5965 | 0,001104 |
| **Media:** | **74,9062** | **0** | **37,4531** | **0,0018474** |
| | Greedy-Relief | | | |
| **Partición 1** | 79,3103 | 37,0968 | 58,2036 | 0,004727 |
| **Partición 2** | 72,4138 | 35,4839 | 53,9488 | 0,004518 |
| **Partición 3** | 80,7018 | 35,4839 | 58,0928 | 0,004508 |
| **Partición 4** | 70,1754 | 38,7097 | 54,4426 | 0,004542 |
| **Partición 5** | 71,9298 | 58,0645 | 64,9972 | 0,004436 |
| **Media:** | **74,9062** | **40,9677** | **57,937** | **0,0045462** |
| | BL | | | |
| **Partición 1** | 67,2414 | 77,4194 | 72,3304 | 7,30235 |
| **Partición 2** | 74,1379 | 80,6452 | 77,3915 | 10,9043 |
| **Partición 3** | 71,9298 | 85,4839 | 78,7068 | 23,0629 |
| **Partición 4** | 70,1754 | 77,4194 | 73,7974 | 7,26972 |
| **Partición 5** | 78,9474 | 80,6452 | 79,7963 | 10,9879 |
| **Media:** | **72,4864** | **80,3226** | **76,4045** | **11,9054** |

Figure 2: Algoritmos k-NN, Relief y Local Search en Colposcopy.csv

| | COLPOSCOPY | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AGG_BLX | | | |
| **Partición 1** | 81,0345 | 75,8065 | 78,4205 | 46,6095 |
| **Partición 2** | 58,6207 | 83,871 | 71,2458 | 42,7991 |
| **Partición 3** | 78,9474 | 80,6452 | 79,7963 | 44,7518 |
| **Partición 4** | 71,9298 | 77,4194 | 74,6746 | 47,9142 |
| **Partición 5** | 71,9298 | 69,3548 | 70,6423 | 50,1035 |
| **Media:** | **72,4924** | **77,4194** | **74,9559** | **46,4356** |
| | AGG_CA | | | |
| **Partición 1** | 77,5862 | 66,129 | 71,8576 | 50,9519 |
| **Partición 2** | 70,6897 | 62,9032 | 66,7964 | 51,5923 |
| **Partición 3** | 82,4561 | 66,129 | 74,2926 | 52,3728 |
| **Partición 4** | 70,1754 | 61,2903 | 65,7329 | 54,2433 |
| **Partición 5** | 66,6667 | 67,7419 | 67,2043 | 51,2676 |
| **Media:** | **73,5148** | **64,8387** | **69,1768** | **52,0856** |
| | AGE_BLX | | | |
| **Partición 1** | 74,1379 | 87,0968 | 80,6174 | 43,6059 |
| **Partición 2** | 74,1379 | 83871 | 79,0044 | 45,2081 |
| **Partición 3** | 82,4561 | 83871 | 83,1636 | 44,1516 |
| **Partición 4** | 68,4211 | 79,0323 | 73,7267 | 47,4391 |
| **Partición 5** | 75,4386 | 88,7097 | 82,0741 | 42,8586 |
| **Media:** | **74,9183** | **84,5161** | **79,7172** | **44,6527** |
| | AGE_CA | | | |
| **Partición 1** | 79,3103 | 74,1935 | 76,7519 | 47,5808 |
| **Partición 2** | 70,6897 | 83,871 | 77,2803 | 48,0552 |
| **Partición 3** | 75,4386 | 83,871 | 79,6548 | 46,6335 |
| **Partición 4** | 66,6667 | 79,0323 | 72,8495 | 48,6645 |
| **Partición 5** | 75,4386 | 83,871 | 79,6548 | 46,0709 |
| **Media:** | **73,5088** | **80,9677** | **77,2383** | **47,401** |

Figure 3: Algoritmos Genéticos en Colposcopy.csv

| | COLPOSCOPY | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AM_BLX(10,1,0) | | | |
| **Partición 1** | 77,5862 | 82,2581 | 79,9221 | 49,0225 |
| **Partición 2** | 72,4138 | 77,4194 | 74,9166 | 49,12 |
| **Partición 3** | 71,9298 | 82,2581 | 77,0939 | 47,2666 |
| **Partición 4** | 73,6842 | 83,871 | 78,7776 | 53,2437 |
| **Partición 5** | 73,6842 | 80,6452 | 77,1647 | 49,4555 |
| **Media:** | **73,8596** | **81,2903** | **77,575** | **49,6217** |
| | AM_BLX(10,0,1) | | | |
| **Partición 1** | 74,1379 | 79,0323 | 76,5851 | 46,8333 |
| **Partición 2** | 81,0345 | 83,871 | 82,4527 | 48,5753 |
| **Partición 3** | 75,4386 | 75,8065 | 75,6225 | 48,6499 |
| **Partición 4** | 64,9123 | 75,8065 | 70,3594 | 46,437 |
| **Partición 5** | 78,9474 | 79,0323 | 78,9898 | 42,5833 |
| **Media:** | **74,8941** | **78,7097** | **76,8019** | **46,6158** |
| | AM_BLX(10,0,1)_mej | | | |
| **Partición 1** | 75,8621 | 74,1935 | 75,0278 | 47,2963 |
| **Partición 2** | 70,6897 | 80,6452 | 75,6674 | 47,5943 |
| **Partición 3** | 84,2105 | 77,4194 | 80,8149 | 45,3372 |
| **Partición 4** | 78,9474 | 80,6452 | 79,7963 | 46,1907 |
| **Partición 5** | 78,9474 | 80,6452 | 79,7963 | 71,1888 |
| **Media:** | **77,7314** | **78,7097** | **78,2205** | **51,5215** |

Figure 4: Algoritmos Meméticos en Colposcopy.csv

| | IONOSPHERE | | | |
|---|---|---|---|---|
| | 1-NN | | | |
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| **Partición 1** | 90,1408 | 0 | 45,0704 | 0,002019 |
| **Partición 2** | 80 | 0 | 40 | 0,001399 |
| **Partición 3** | 82,8571 | 0 | 41,4286 | 0,001052 |
| **Partición 4** | 92,8571 | 0 | 46,4286 | 0,000906 |
| **Partición 5** | 87,1429 | 0 | 43,5714 | 0,000898 |
| **Media:** | **86,5996** | **0** | **43,2998** | **0,0012548** |
| | **Greedy-Relief** | | | |
| **Partición 1** | 90,1408 | 2,94118 | 46,541 | 0,003929 |
| **Partición 2** | 81,4286 | 2,94118 | 42,1849 | 0,004313 |
| **Partición 3** | 82,8571 | 2,94118 | 42,8992 | 0,003928 |
| **Partición 4** | 92,8571 | 2,94118 | 47,8992 | 0,003958 |
| **Partición 5** | 90 | 2,94118 | 46,4706 | 0,004275 |
| **Media:** | **87,4567** | **2,94118** | **45,199** | **0,0040806** |
| | **BL** | | | |
| **Partición 1** | 91,5493 | 85,2941 | 88,4217 | 4,2856 |
| **Partición 2** | 80 | 88,2353 | 84,1176 | 4,44632 |
| **Partición 3** | 87,1429 | 88,2353 | 87,6891 | 4,8549 |
| **Partición 4** | 95,7143 | 88,2353 | 91,9748 | 5,75943 |
| **Partición 5** | 85,7143 | 91,1765 | 88,4454 | 3,69103 |
| **Media:** | **88,0241** | **88,2353** | **88,1297** | **4,60746** |

Figure 5: Algoritmos k-NN, Relief y Local Search en Ionosphere.csv

| | IONOSPHERE | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AGG_BLX | | | |
| **Partición 1** | 84,507 | 91,1765 | 87,8418 | 35,0041 |
| **Partición 2** | 87,1429 | 82,3529 | 84,7479 | 42,0527 |
| **Partición 3** | 82,8571 | 85,2941 | 84,0756 | 37,9694 |
| **Partición 4** | 88,5714 | 85,2941 | 86,9328 | 37,763 |
| **Partición 5** | 88,5714 | 82,3529 | 85,4622 | 36,9627 |
| **Media:** | **86,33** | **85,2941** | **85812** | **37,9504** |
| | AGG_CA | | | |
| **Partición 1** | 83,0986 | 79,4118 | 81,2552 | 41,3409 |
| **Partición 2** | 82,8571 | 70,5882 | 76,7227 | 42,0788 |
| **Partición 3** | 80 | 70,5882 | 75,2941 | 42,78 |
| **Partición 4** | 85,7143 | 70,5882 | 78,1513 | 45,1645 |
| **Partición 5** | 91,4286 | 76,4706 | 83,9496 | 42,6796 |
| **Media:** | **84,6197** | **73,5294** | **79,0746** | **42,8088** |
| | AGE_BLX | | | |
| **Partición 1** | 81,6901 | 91,1765 | 86,4333 | 34,4981 |
| **Partición 2** | 81,4286 | 91,1765 | 86,3025 | 41624 |
| **Partición 3** | 84,2857 | 88,2353 | 86,2605 | 36,8601 |
| **Partición 4** | 91,4286 | 88,2353 | 89,8319 | 37,2436 |
| **Partición 5** | 87,1429 | 91,1765 | 89,1597 | 42,2954 |
| **Media:** | **85,1952** | **90** | **87,5976** | **38,5043** |
| | AGE_CA | | | |
| **Partición 1** | 87,3239 | 91,1765 | 89,2502 | 36,0913 |
| **Partición 2** | 85,7143 | 91,1765 | 88,4454 | 37,6856 |
| **Partición 3** | 81,4286 | 79,4118 | 80,4202 | 35,9104 |
| **Partición 4** | 87,1429 | 88,2353 | 87,6891 | 40,0104 |
| **Partición 5** | 90 | 88,2353 | 89,1176 | 37,1055 |
| **Media:** | **86,3219** | **87,6471** | **86,9845** | **37,3606** |

Figure 6: Algoritmos Genéticos en Ionosphere.csv

| | IONOSPHERE | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AM_BLX(10,1,0) | | | |
| **Partición 1** | 84,507 | 88,2353 | 86,3712 | 35,446 |
| **Partición 2** | 82,8571 | 91,1765 | 87,0168 | 38,2486 |
| **Partición 3** | 85,7143 | 91,1765 | 88,4454 | 34,7838 |
| **Partición 4** | 82,8571 | 88,2353 | 85,5462 | 39,2386 |
| **Partición 5** | 87,1429 | 88,2353 | 87,6891 | 43,3305 |
| **Media:** | **84,6157** | **89,4118** | **87,0137** | **38,2095** |
| | AM_BLX(10,0,1) | | | |
| **Partición 1** | 83,0986 | 85,2941 | 84,1964 | 36,2763 |
| **Partición 2** | 81,4286 | 91,1765 | 86,3025 | 42,1 |
| **Partición 3** | 82,8571 | 88,2353 | 85,5462 | 36,0671 |
| **Partición 4** | 82,8571 | 91,1765 | 87,0168 | 36,5171 |
| **Partición 5** | 87,1429 | 91,1765 | 89,1597 | 38,4302 |
| **Media:** | **83,4769** | **89,4118** | **86,4443** | **37,8782** |
| | AM_BLX(10,0,1)_mej | | | |
| **Partición 1** | 84,507 | 88,2353 | 86,3712 | 34,7658 |
| **Partición 2** | 90 | 91,1765 | 90,5882 | 35,1409 |
| **Partición 3** | 92,8571 | 91,1765 | 92,0168 | 40,8443 |
| **Partición 4** | 81,4286 | 91,1765 | 86,3025 | 49,3906 |
| **Partición 5** | 87,1429 | 88,2353 | 87,6891 | 59,5511 |
| **Media:** | **87,1871** | **90** | **88,5936** | **43,9385** |

Figure 7: Algoritmos Meméticos en Ionosphere.csv

| | TEXTURE | | | |
|---|---|---|---|---|
| | 1-NN | | | |
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| **Partición 1** | 93,6364 | 0 | 46,8182 | 0,005242 |
| **Partición 2** | 89,0909 | 0 | 44,5455 | 0,004109 |
| **Partición 3** | 94,5455 | 0 | 47,2727 | 0,003104 |
| **Partición 4** | 92,7273 | 0 | 46,3636 | 0,002719 |
| **Partición 5** | 92,7273 | 0 | 46,3636 | 0,002923 |
| **Media:** | **92,5455** | **0** | **46,2727** | **0,0036194** |
| | Greedy-Relief | | | |
| **Partición 1** | 91,8182 | 15 | 53,4091 | 0,011238 |
| **Partición 2** | 91,8182 | 2,5 | 47,1591 | 0,012252 |
| **Partición 3** | 95,4545 | 2,5 | 48,9773 | 0,012533 |
| **Partición 4** | 92,7273 | 2,5 | 47,6136 | 0,011493 |
| **Partición 5** | 93,6364 | 5 | 49,3182 | 0,01229 |
| **Media:** | **93,0909** | **5,5** | **49,2955** | **0,0119612** |
| | BL | | | |
| **Partición 1** | 81,8182 | 85 | 83,4091 | 15,0112 |
| **Partición 2** | 85,4545 | 85 | 85,2273 | 14,3572 |
| **Partición 3** | 87,2727 | 82,5 | 84,8864 | 11,6376 |
| **Partición 4** | 90 | 85 | 87,5 | 19,6986 |
| **Partición 5** | 91,8182 | 85 | 88,4091 | 16,9499 |
| **Media:** | **87,2727** | **84,5** | **85,8864** | **15,5309** |

Figure 8: Algoritmos k-NN, Relief y Local Search en Texture.csv

| | TEXTURE | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AGG_BLX | | | |
| **Partición 1** | 91,8182 | 80 | 85,9091 | 109,924 |
| **Partición 2** | 89,0909 | 80 | 84,5455 | 106,906 |
| **Partición 3** | 90,9091 | 80 | 85,4545 | 115,347 |
| **Partición 4** | 90 | 77,5 | 83,75 | 114,734 |
| **Partición 5** | 89,0909 | 77,5 | 83,2955 | 112,521 |
| **Media:** | **90,1818** | **79** | **84,5909** | **111,886** |
| | AGG_CA | | | |
| **Partición 1** | 90 | 80 | 85 | 120,377 |
| **Partición 2** | 93,6364 | 72,5 | 83,0682 | 126,38 |
| **Partición 3** | 93,6364 | 72,5 | 83,0682 | 123,135 |
| **Partición 4** | 87,2727 | 70 | 78,6364 | 127,926 |
| **Partición 5** | 87,2727 | 70 | 78,6364 | 124,747 |
| **Media:** | **90,3636** | **73** | **81,6818** | **124,513** |
| | AGE_BLX | | | |
| **Partición 1** | 91,8182 | 82,5 | 87,1591 | 105,912 |
| **Partición 2** | 90 | 85 | 87,5 | 107,17 |
| **Partición 3** | 90,9091 | 87,5 | 89,2045 | 104,692 |
| **Partición 4** | 90 | 87,5 | 88,75 | 105,887 |
| **Partición 5** | 87,2727 | 85 | 86,1364 | 106,347 |
| **Media:** | **90** | **85,5** | **87,75** | **106,001** |
| | AGE_CA | | | |
| **Partición 1** | 88,1818 | 87,5 | 87,8409 | 110,778 |
| **Partición 2** | 88,1818 | 82,5 | 85,3409 | 131,699 |
| **Partición 3** | 90,9091 | 87,5 | 89,2045 | 108,363 |
| **Partición 4** | 90,9091 | 82,5 | 86,7045 | 110,064 |
| **Partición 5** | 93,6364 | 85 | 89,3182 | 107,941 |
| **Media:** | **90,3636** | **85** | **87,6818** | **113,769** |

Figure 9: Algoritmos Genéticos en Texture.csv

| | TEXTURE | | | |
|---|---|---|---|---|
| | %tasa_clas | %tasa_red | Agregado | T(seg) |
| | AM_BLX(10,1,0) | | | |
| **Partición 1** | 90,9091 | 85 | 87,9545 | 106,946 |
| **Partición 2** | 92,7273 | 87,5 | 90,1136 | 102,27 |
| **Partición 3** | 89,0909 | 87,5 | 88,2955 | 103,049 |
| **Partición 4** | 87,2727 | 85 | 86,1364 | 106,731 |
| **Partición 5** | 87,2727 | 87,5 | 87,3864 | 109,326 |
| **Media:** | **89,4545** | **86,5** | **87,9773** | **105,664** |
| | AM_BLX(10,0,1) | | | |
| **Partición 1** | 88,1818 | 85 | 86,5909 | 103,143 |
| **Partición 2** | 89,0909 | 82,5 | 85,7955 | 105,262 |
| **Partición 3** | 88,1818 | 82,5 | 85,3409 | 102,398 |
| **Partición 4** | 86,3636 | 85 | 85,6818 | 101,424 |
| **Partición 5** | 90,9091 | 87,5 | 89,2045 | 99,2087 |
| **Media:** | **88,5455** | **84,5** | **86,5227** | **102,287** |
| | AM_BLX(10,0,1)_mej | | | |
| **Partición 1** | 91,8182 | 87,5 | 89,6591 | 101,405 |
| **Partición 2** | 85,4545 | 87,5 | 86,4773 | 103,054 |
| **Partición 3** | 96,3636 | 87,5 | 91,9318 | 99,5878 |
| **Partición 4** | 85,4545 | 85 | 85,2273 | 158,162 |
| **Partición 5** | 89,0909 | 85 | 87,0455 | 142,022 |
| **Media:** | **89,6364** | **86,5** | **88,0682** | **120,846** |

Figure 10: Algoritmos Meméticos en Texture.csv

Vemos que en general, de media todos los algoritmos evolutivos tardan tiempos similares en ejecutar los datos. A mayor cantidad de datos, más tardan, por lo que la ejecución de los algoritmos evolutivos sobre el conjunto de datos *texture.csv* son los que más tardan en media.

Como ya hemos visto antes, para conjuntos de datos pequeños como *ionosphere.csv* no compensa utilizar algoritmos evolutivos, ya que el tiempo de ejecución comparado con el de **BL** es bastante mayor. Mientras que para el conjunto *texture.csv* el algoritmo **Memético (10,0.1) mejor** es el que obtiene mejores resultados.

### 6.1. Sobreaprendizaje

Decimos que un algoritmo ha sido sobreentrenado, si el algoritmo empleado sobre un conjunto de datos es mejor sobre los datos de entrenamiento que sobre los de prueba. Esto es malo ya que el algoritmo puede ser bueno para un conjunto de datos específico pero perder generalidad para datos fuera del conjunto de entrenamiento.

Para ver que algoritmos sufren de este problema y por tanto son peores a la larga, primero entrenaremos un conjunto de datos y más tarde compararemos el resultado de evaluar ese conjunto de datos sobre él mismo con el de evaluarlo sobre otro conjunto de prueba.

Solo tenemos que hacer la siguiente modificación en las funciones de *ejecutarAlgoritmo*.

```
function ejecutarAlgoritmo(particion, num_part, SEED)
  ...
  start_timers();
  AGE_CA(train, w);
  results = KNN_LOO(train, train, w)
  results.tiempo = elapsed_time(REAL)
  ...
  return results
end function
```

Para no abusar del número de tablas incluidas, haremos las comparaciones sobre la tabla global de ambas ejecuciones.

| | COLPOSCOPY | | IONOSPHERE | | TEXTURE | |
|---|---|---|---|---|---|---|
| | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red |
| **1-NN** | 75,6085 | 0 | 86,7517 | 0 | 92,5455 | 0 |
| **Relief** | 78,6588 | 40,9677 | 88,0336 | 2,94118 | 94,1364 | 5,5 |
| **BL** | 82,4006 | 81,2903 | 92,0216 | 88,2353 | 92,0909 | 84,5 |
| **AGG_BLX** | 80,6607 | 77,4194 | 89,8129 | 85,8824 | 92,2727 | 80 |
| **AGG_CA** | 77,4384 | 64,8387 | 88,8904 | 70 | 92,2727 | 72 |
| **AGE_BLX** | 82,2301 | 84,8387 | 90,0991 | 88,8235 | 91,8636 | 86 |
| **AGE_CA** | 80,4006 | 80,6452 | 92,0933 | 89,4118 | 92,2273 | 85 |
| **AM(1.0)** | 83,7103 | 84,8387 | 91,88 | 90 | 92,7727 | 87,5 |
| **AM(0.1)** | 83,8834 | 78,7097 | 92,5214 | 90 | 92,4091 | 84,5 |
| **AM(0.1)mej** | 82,8369 | 79,0323 | 91,0989 | 89,4118 | 93,2727 | 86,5 |

Figure 11: Tabla datos comparativa train-train

| | COLPOSCOPY | | IONOSPHERE | | TEXTURE | |
|---|---|---|---|---|---|---|
| | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red | %tasa_clas | %tasa_red |
| **1-NN** | 74,9062 | 0 | 86,5996 | 0 | 92,5455 | 0 |
| **Relief** | 74,9062 | 40,9677 | 87,4567 | 2,94118 | 93,0909 | 5,5 |
| **BL** | 72,4864 | 80,3226 | 88,0241 | 88,2353 | 87,2727 | 84,5 |
| **AGG_BLX** | 72,4924 | 77,4194 | 86,33 | 85,2941 | 90,1818 | 79 |
| **AGG_CA** | 73,5148 | 64,8387 | 84,6197 | 73,5294 | 90,3636 | 73 |
| **AGE_BLX** | 74,9183 | 84,5161 | 85,1952 | 90 | 90 | 85,5 |
| **AGE_CA** | 73,5088 | 80,9677 | 86,3219 | 87,6471 | 90,3636 | 85 |
| **AM(1.0)** | 73,8596 | 81,2903 | 84,6157 | 89,4118 | 89,4545 | 86,5 |
| **AM(0.1)** | 74,8941 | 78,7097 | 83,4769 | 89,4118 | 88,5455 | 84,5 |
| **AM(0.1)mej** | 77,7314 | 78,7097 | 87,1871 | 90 | 89,6364 | 86,5 |

Figure 12: Tabla datos comparativa train-test

Como podemos ver, las tasas de las ejecuciones train-train son en media más altas que las de train-test. Esto se debe a que todos los algoritmos en mayor o menor medida sobreentrenan los conjuntos de datos, es decir, se especializan más en el conjunto sobre el que entrenan.

Además, podemos notar que los algoritmos evolutivos para conjuntos de datos pequeños, son los que más sobreentrenan, ya que obtienen muy buenas tasas de clasificación sobre el conjunto train y pierden en calidad a la hora de clasificar los del conjunto test.

## 7. Bibliografía

[1] Guión de Prácticas y Seminarios de la Asignatura de Metaheurísticas.